

**Richiami di C++ / C**  
**Insegnamento di Sistemi di Elaborazione**  
**Corso di L.M. In Ing. Elettronica**

Ing. Simone Brienza

E-mail: [simone.brienza@for.unipi.it](mailto:simone.brienza@for.unipi.it)

Pervasive Computing & Networking Lab (PerLab) <http://www.perlab.it>

Dipartimento di Ingegneria dell'Informazione, Università di Pisa

# Tipi di Dato

# Tipi di dato: char



- `char`
  - occupano generalmente **1** byte.
  - Sono usati tipicamente per rappresentare un carattere ASCII o un byte di memoria. Più raramente vengono impiegati per la rappresentazione di piccoli interi con o senza segno.
  - **-128..0..127** per signed char
  - **0..255** per unsigned char

# Tipi di dato: `int`



- `int`
  - Occupano generalmente 2 o 4 byte a seconda dell'implementazione del compilatore e dell'architettura del sistema (16 o 32 bit).
  - E' la rappresentazione tipica per numeri interi con o senza segno.
  - Si possono impiegare i termini `short` o `long` per forzare l'intero alla minima o massima dimensione ammessa.
  - Possono essere utilizzate le parole chiave `signed` e `unsigned` per specificare se il tipo sia dotato o no di segno.

# Tipi di dato : float e double



- float e double
  - I float occupano 4 byte di memoria, i double ne occupano 8.
  - Rappresentano numeri reali in singola precisione (float) o in doppia precisione (double).
  - Molti compilatori supportano anche il tipo long double per la rappresentazione di un numero reale in quadrupla precisione.
  - Si differenziano per i bit che sono riservati per la loro rappresentazione, non solo per il range di rappresentazione, ma anche per il numero di cifre dopo la virgola che possono essere rappresentate

# Tipi di dato : `void`



- `void`
  - Questo tipo specifica un insieme vuoto di valori.
  - Viene utilizzato nella dichiarazione e definizione di funzioni come le funzioni che non ritornano valori, oppure che non accettano parametri

- **Operatori aritmetici:**
  - $a = b$ ,  $a + b$ ,  $a - b$ ,  $++a$ ,  $a++$ ,  $--a$ ,  $a--$ ,  $a * b$ ,  $a / b$ ,  $a \% b$
- **Operatori di confronto:**
  - $a == b$ ,  $a != b$ ,  $a > b$ ,  $a >= b$ ,  $a < b$ ,  $a <= b$
- **Operatori logici:**
  - $!a$ ,  $a \&\& b$ ,  $a || b$
- **Operatori bit a bit:**
  - $\sim a$ ,  $a \& b$ ,  $a | b$ ,  $a \wedge b$ ,  $a \ll b$ ,  $a \gg b$

# Istruzioni condizionali



- **if** (*expr*)  
    *statement*;  
**else**  
    *statement*;
- (*expr*) ? *statement* : *statement*;
- **switch** (*variable*) {  
    **case 1:**  
        *statement*;  
        **break**;  
    **case 2:**  
        ...  
}



# Istruzioni iterative



- **while** (*expr*) {  
    *statement*;  
    ...  
}
- **do** {  
    *statement*;  
    ...  
} **while** (*expr*);
- **for** (*initialization*; *condition*; *increase*) {  
    *statement*;  
    ...  
}

es: for (int i = 0; i < 10; i++) {  
    cout << i << ", ";  
}

```
tipo identificatore[numero_elementi];
```

- **definisco ed inizializzo un array di 4 interi**

```
int a1[4]={10, 20, 30, 40};
```

- **equivalente alla definizione di a1**

```
int a2[]={10, 20, 30, 40};
```

- **definisco un'array di 30 interi, ma ne inizializzo solamente 4**

```
int a3[30]={10, 20, 30, 40};
```

- **definisco una stringa costruita con un array di 5 caratteri ed inizializzata con "Ciao"**

```
char s1[5]={'C', 'i', 'a', 'o', '\0'};
```

- **equivalente a s1**

```
char s2[]="Ciao";
```

- **Per potere accedere ad un elemento dell'array, si utilizza un indice variabile da 0 (per riferirsi al primo elemento) alla dimensione-1 (per riferirsi all'ultimo elemento).**

# Array multidimensionali



```
tipo identificatore [dimensione1] [dimensione2] ...;
```

- La memorizzazione avviene per righe, corrispondenti al primo indice.
- Per inizializzare un array multidimensionale, si nidificano le inizializzazioni all'interno delle parentesi graffe; le inizializzazioni avvengono per righe.
- In fase di definizione dell'array è anche possibile non indicare la prima dimensione, lasciando il compito al compilatore di ricavare il numero delle righe in base alle inizializzazioni effettuate. Le altre dimensioni devono essere specificate.

```
int nome_array [10] [2];
```

```
int a1 [3][4]={{ 10, 20, 30, 40},{ 50, 60, 70, 80},{ 90, 100, 110, 120}};  
int a1 [][4]={{ 10, 20, 30, 40},{ 50, 60, 70, 80},{ 90, 100, 110, 120}};
```

# Struct



```
struct { listaDichiarazioneCampi } nomeVarStrutturale...;
```

- **Struttura di dati correlati fra loro**
- **Per accedere ai vari campi della struttura si impiega l'operatore . (punto)**
- **Una struttura può essere inizializzata direttamente in fase di definizione fornendo fra parentesi graffe la lista dei valori che si intende assegnare ai vari campi.**

```
struct data {      /* dichiarazione struttura data */
    int gg;        /* giorno */
    int mm;        /* mese */
    int aa;        /* anno */
};

struct data data1, data2;          /* definizione delle variabili */
struct data data3 = {1, 1, 2000}; /* Inizializzazione contestuale */
data1.gg = data3.gg;              /* copio i singoli campi */
data1.mm = data3.mm;
data1.aa = 2001;
data2 = data3;                    /* assegnamenti fra strutture */
```

```
tipo *varPuntatore;
```

- Il puntatore contiene l'indirizzo di memoria dell'area a cui punta
- Per allocare dinamicamente memoria nello heap, usualmente viene impiegata la funzione di libreria

```
void* malloc(int size).
```

- L'accesso alla memoria tramite il puntatore viene effettuato per mezzo dell'operatore di indirizzione \* (asterisco)
- Un puntatore inizializzato a `NULL`, per definizione non punta ad un indirizzo di memoria valido.

```
int *ptrInt; /* definisce ptrInt come puntatore a intero */
```

# Operatore indirizzo &



- Ad un puntatore può essere assegnato direttamente l'indirizzo dell'area di memoria riferita ad una variabile per mezzo dell'operatore indirizzo &

```
int a=5;
```

```
int *p;
```

```
p=&a;
```

← Cosa provoca?

```
*p=6;
```

← Cosa provoca?

# Funzioni (1 di 2)



- **Definizione**

```
tiporitornato
nomefunzione(dichiarazione_parametri_formali){
    definizioni/dichiarazioni
    lista_statement
}
```

- **Condizioni di terminazione della funzione:**
  - **esecuzione dell'istruzione** `return`
  - **l'ultima istruzione del corpo della funzione è stata eseguita**
- **Quando la funzione termina, il controllo torna al chiamante.**

# Funzioni (2 di 2)



- **Invocazione**

```
nomefunzione(lista_parametri_attuali)
```

- **Dichiarazione**

```
tiporisultato  
nomefunzione(dichiarazione_parametri_formali  
);
```

- **La dichiarazione della funzione è necessaria se viene invocata prima della sua definizione, oppure se è definita in altro file e compilata separatamente.**



# Passaggio di parametri (1 di 2)



- Nel C il passaggio dei parametri ad una funzione, avviene **sempre per valore**, per cui non è possibile che una funzione abbia degli effetti collaterali sui parametri passati.

```
#include <stdio.h>
#include <stdlib.h>
```

```
void funz(int a);
int main(int argc, char *argv) {
    int a=10;
    printf("a = %d\n\n", a); /* stampa a = 10*/
    funz(a);
    printf("a = %d\n\n", a); /* stampa a = 10*/
    return 0;
}
```

```
void funz(int a) {
    a/=2;
    printf("a = %d\n\n", a); /* stampa a = 5*/
}
```

# Passaggio di parametri (2 di 2)



- Qualora, invece, sia necessario che la chiamata di una funzione produca degli effetti collaterali, allora diventa indispensabile passare alla funzione gli **indirizzi di memoria** di dove si devono ottenere gli effetti collaterali.
- In questo modo, i puntatori sono ancora passati per valore, e pertanto non possono essere modificati dalla funzione, mentre può venire modificato il contenuto a cui essi puntano.

```
int main(void) {  
    int a1, b1, c1;  
    inizializza(&a1, &b1, &c1); /*chiamata alla funzione*/  
}
```

```
void inizializza(int *a, int *b, int *c) {  
    *a = 5;  
    *b = 7;  
    *c = 20;  
}
```

# Funzione `main()`



- La funzione `main()` deve sempre essere presente in un programma C ed è la prima funzione a cui viene passato il controllo.
- A sua volta la funzione `main()` può invocare altre funzioni.

```
int main(int argc, char *argv[]) {...}
```
- `argc` contiene il numero dei parametri sulla riga di comando, incluso il nome del programma. Pertanto `argc` è sempre  $\geq 1$ .
- `argv` è un array di puntatori a `char`. Ogni puntatore riferenzia un singolo parametro della linea di comando sotto forma sempre di stringa. I parametri inseriti nella linea di comando vengono sempre acquisiti dal programma come stringhe, anche nel caso che questi rappresentino dei valori numerici.
  - `argv[0]` è il nome del programma ed è sempre presente.
  - `argv[1]` è il primo parametro (se esistente) digitato dopo il nome del programma.
  - `argv[argc-1]` è l'ultimo parametro passato al programma.

# Funzione `main()`



- Per tradurre una stringa in un intero potete usare la funzione

```
int atoi(char*)
```

che prende in ingresso una stringa e restituisce l'intero corrispondente all'intero immesso sotto forma di stringa.

- E' utile quando si vuole passare parametri numerici al `main()`.

# Differenze C/C++

# Definizione variabili



- Le variabili possono essere definite solo all'inizio di un blocco

## Stile C++

```
int main (void) {  
int a=5, b;  
a=func(1000);  
int b=f(a);  
...  
for (int i=0; a<100; i++ ) {  
b=f(a);  
int c=0;  
...  
}  
}
```

## Stile C

```
int main (void) {  
int a=5, i, b;  
int b=f(a);  
a=func(1000);  
...  
for ( i=0; a<100; i++ ) {  
    int c=0;  
b=f(a);  
...  
}  
}
```

- **Le strutture vanno sempre riferite con la parola chiave `struct`**

## Stile C++

```
struct Complesso{  
double re;  
double im;  
}
```

```
int main (void){  
int a=5;  
Complesso c;  
}
```

## Stile C

```
struct Complesso{  
double re;  
double im;  
}
```

```
int main (void){  
int a=5;  
struct Complesso c;  
}
```

# Allocazione della memoria dinamica



```
#include <stdlib.h>
int main (void){
int mem_size=5;
void *ptr;
ptr = malloc(mem_size) ;
if(ptr == NULL) {
/*errore*/
}
...
}
```

- **malloc()** accetta come parametro la quantità di memoria da allocare. In caso di successo viene tornato l'indirizzo dell'area allocata. In caso di fallimento (es. memoria insufficiente) torna il valore `NULL` che indica un errore che deve essere gestito
- L'indirizzo tornato dalla `malloc()` deve venire assegnato alla variabile puntatore per potere accedere alla memoria allocata.
- `mem_size` : numero di bytes che si vuole allocare nello heap
- `man malloc` : mostra la documentazione



# Deallocazione della memoria dinamica



```
#include <stdlib.h>
int main (void){
int mem_size=5;
void *ptr;
ptr = malloc(mem_size);
if(ptr == NULL) {
/*errore*/
}
...
free(ptr) ;
}
```

- **Per liberare la memoria precedentemente allocata, viene chiamata la funzione `free()` della libreria standard, avente come argomento il valore del puntatore dell'area di memoria da deallocare.**

# Output su std output (video)



```
#include <stdlib.h>
int main(void) {
    int i=5;
    char *str="ciao ciao\n";
    printf(str);
    printf("i=%d\n", i);
    ...
}
```

```
>./prog
ciao ciao
i=5
```

# Sequenze di Escape



- `\b`      **backspace**
- `\f`      **FF - form feed (salto pagina)**
- `\n`      **LF - line feed o newline**
- `\r`      **CR - carriage return (ritorno del carrello)**
- `\t`      **tabulazione orizzontale**
- `\v`      **tabulazione verticale**
- `\\`      **\ (barra inversa)**
- `\?`      **? (punto interrogativo)**
- `\'`      **' (apice)**
- `\"`      **" (doppi apici)**
- `\0`      **NULL (carattere nullo)**

# Specificatori di formato



- `% [flags] [width] [.precision] [length] specifier`

<i>specifier</i>	<b>Output</b>	<b>Example</b>
<b>d</b>	<b>Signed decimal integer</b>	<b>392</b>
<b>u</b>	<b>Unsigned decimal integer</b>	<b>7235</b>
<b>X</b>	<b>Unsigned hexadecimal integer (uppercase)</b>	<b>7FA</b>
<b>f</b>	<b>Decimal floating point,</b>	<b>392.65</b>
<b>c</b>	<b>Character</b>	<b>a</b>
<b>s</b>	<b>String of characters</b>	<b>sample</b>
<b>p</b>	<b>Pointer address</b>	<b>b8000000</b>

- **Es:**

```
printf ("test: %10d \n", 1977); // test: 1977
printf ("test: %+10d \n", 1977); // test: +1977
printf ("test: %4.2f \n", 3.1416); // test: 3.14
```

# Input da std input (tastiera):

`fgets()`



```
#include <stdlib.h>
int main (void){
char *str = (char *) malloc(100);
fgets(str, 100, stdin);
...
}
```

- `str` : array di caratteri
- `stdin` : standard input
- Prende 99 caratteri e termina con `\0`
- Si può usare anche la funzione `scanf()` per prelevare dati da riga di comando

```
#include <stdlib.h>
int main (void){
int i;
scanf("%d", &i);
...
}
```

# Lunghezza stringhe



```
#include<string.h>

int main (void){
char *str1 = "ciao ciao\n";
char *str2 = malloc(10);
int len, len1, len2;
...
len  = strlen(str1); //10 ma 11 byte allocati in memoria
len1 = sizeof(str1); //4
len2 = sizeof(str2); //4
...
}
```

# Confronto stringhe



```
#include<string.h>

int main (void){
char *str1 = "ciao", str2="bye";
int i = strcmp(str1,str2);
...
}
```

- $i < 0$  str1 alfabeticamente minore di str2
- $i > 0$  str1 alfabeticamente maggiore di str2
- $i = 0$  str1 uguale a str2
  
- Case sensitive

# Copia stringhe



- `strcpy` : copia la stringa passata come secondo parametro nel primo parametro, senza fare controlli sulla lunghezza.
- `strncpy` : prende come terzo parametro il massimo numero di caratteri. La parte vuota viene riempita con `\0`.

```
#include<string.h>
int main (void){
char str1 [100];
strncpy(str1, "ciao\n", sizeof(str1)-1);
...
}
```



# Apertura file



```
#include<stdio.h>
int main (void){
    FILE *fp;
    fp = fopen("/tmp/prova.txt", "r");
    if (fp == NULL) {
        /*errore*/
    }
    ...
}
```

- **FILE \*** : struttura dati per gestire le operazioni sui file
- Quando un file viene aperto si ha un puntatore al primo byte del file
- Tale puntatore dice da dove parte la prossima operazione di I/O
- **Modalità di apertura**
  - **r** : read-only
  - **w** : write-only
  - **r+** : read and write
  - **a** : append
  - **a+** : append and read

# Lettura da file binario



```
#include <stdio.h>
int main (void) {
int ret;
char str[1024];
FILE *fp;
fp=fopen("/tmp/prova.txt","r");
ret = fread(str,1,sizeof(str)-1,fp);
}
```

1. **Buffer di immagazzinamento**
  2. **Dimensione degli elementi da leggere (in bytes)**
  3. **Numero di elementi da leggere**
  4. **File da cui leggere**
- **Se il file è più corto torna errore altrimenti torna il numero di elementi letti**

# Letture formattata da file



```
int main (void) {  
    int ret, n;  
    FILE *fp;  
    fp=fopen("/tmp/prova.txt","r");  
    ret=fscanf(fp, "%d", &n);  
}
```

# End of File



```
...
ret = fscanf(fp, "%c", &c);
if (ret == EOF) {
    // Sopraggiunta fine del file
}
```

Oppure

```
...
ret = fscanf(fp, "%c", &c);
if (feof(fp)) {
    // Sopraggiunta fine del file
}
```

# Scrittura su file binario



```
#include <stdio.h>
int main (void){
int ret;
char *str = "ciao ciao";
FILE *fp;
fp=fopen("/tmp/prova.txt","w");
ret = fwrite(str,1,strlen(str),fp);
}
```

- 1. puntatore di qualunque tipo, che punta all'area di memoria di ciò che si vuole scrivere sul file**
  - 2. Grandezza (in byte) di ciascun dato da copiare**
  - 3. Numero di elementi da copiare**
  - 4. file su cui scrivere**
- Il valore di ritorno, in caso di successo, indica il numero di elementi scritti su file, ovvero lo stesso numero indicato come terzo parametro.**

# Scrittura formattata su file



```
int main (void) {  
    int ret;  
    char *str = "ciao ciao\n";  
    FILE *fp;  
    fp=fopen("/tmp/prova.txt","w");  
    ret = fprintf(fp,"%s",str);  
}
```

# Dimensione file



```
#include <stdio.h>
#include <sys/stat.h> ← Includere libreria
int main (void) {
int ret, size;
struct stat info;
stat("/tmp/prova.txt", &info);
size = info.st_size;
}
```

# Chiusura file



```
int main (void) {  
FILE *fp;  
fp = fopen("/tmp/prova.txt", "r");  
fclose(fp);  
}
```



# Esecuzione su shell da programma C



```
int main (void) {  
char *str = "/bin/ls -l > file1";  
system(str);  
system("/bin/ls -l > file1");  
}
```

- La `system()` lancia una shell il cui risultato viene rediretto su un file
- Le due chiamate producono lo stesso risultato
- Lo standard output va reindirizzato su un file o viene perso

## ■ Header da includere

- `#include <stdlib.h>` : **malloc(), free(), system()**
- `#include <stdio.h>` : **printf(), fgets(), fopen(), fclose(), fread(), fwrite()**
- `#include <string.h>` : **strlen(), strncpy(), strncat(), strcmp()**
- `#include <sys/types.h>` : **chiamate di sistema per informazioni sui file e sul tempo**
- `#include <sys/stat.h>` : **stat(), fstat()**
- `#include <unistd.h>` : **accesso non bufferizzato ai file, std input, std output, std error.**

# **Comandi per la compilazione**

# File sorgente



- Il file sorgente è identificato dall'estensione `.c`  
(`nomefile.c`)
- `gcc` è il compilatore GNU per programmi scritti in C o C++

# Compilazione di un unico file sorgente



- L'uso tipico del comando `gcc` per compilare programmi in linguaggio C è il seguente:

```
gcc myprog.c -o myprog
```

- Questo comando crea l'eseguibile `myprog` a partire dal sorgente `myprog.c`
- L'opzione `-o` sta per 'output' e se non viene specificata l'eseguibile viene creato col nome `a.out`
- La compilazione vera e propria produce anche il file oggetto `myprog.o` che poi viene automaticamente linkato alle librerie standard per formare l'eseguibile.

# Compilazione di più file separati



- Se il programma è diviso in più file sorgenti è sufficiente elencarli di seguito:

```
gcc mainprog.c baseprog.c commonprog.c -o prog
```

- Oltre all'eseguibile `prog` vengono creati tre file oggetto.

```
gcc -c commonprog.c  
gcc mainprog.o baseprog.o commonprog.o -o prog
```

- Usualmente per programmi con più di un file sorgente si utilizza un Makefile per automatizzare questa procedura (`man make`).

# Opzioni



- -o
  - per specificare il nome del file in cui va salvato il risultato della compilazione.
- -c
  - non fare il linking, ma solo la compilazione.
- -g
  - attiva i simboli di debug. È necessario dare questa opzione al compilatore se si ha intenzione di usare un debugger (`gdb prog`)
- -l
  - serve per aggiungere una libreria .
- -I
  - dice al compilatore di cercare anche nella directory specificata per eventuali file inclusi tramite `#include`
- -Wall
  - abilita la visualizzazione di tutti i messaggi di warning

- Per eseguire il programma è necessario digitare il nome del file eseguibile **preceduto dal suo percorso**

```
./nome_eseguibile
```



**Fine**



Grazie per l'attenzione.